Science of Computer Programming 74 (2009) 261–278

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# Featherweight Java with dynamic and static overloading☆

Lorenzo Bettini [a,*], Sara Capecchi [a], Betti Venneri [b]

[a] Dipartimento di Informatica, Università di Torino, Italy

[b] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy

ARTICLE INFO

*Article history:*
Available online 8 February 2009

*Keywords:*
Object-oriented languages
Featherweight Java
Multi-methods
Static overloading
Dynamic overloading
Type system

ABSTRACT

We propose FMJ (Featherweight Multi Java), an extension of Featherweight Java with encapsulated multi-methods thus providing dynamic overloading. Multi-methods (collections of overloaded methods associated to the same message, whose selection takes place dynamically instead of statically as in standard overloading) are a useful and flexible mechanism which enhances re-usability and separation of responsibilities. However, many mainstream languages, such as, e.g., Java, do not provide it, resorting to only static overloading.

The proposed extension is conservative and type safe: both "message-not-understood" and "message-ambiguous" are statically ruled out. Possible ambiguities are checked during type checking only on method invocation expressions, without requiring to inspect all the classes of a program. A static annotation with type information guarantees that in a well-typed program no ambiguity can arise at run-time. This annotation mechanism also permits modeling static overloading in a smooth way.

Our core language can be used as the formal basis for an actual implementation of dynamic (and static) overloading in Java-like languages.



## 1. Introduction

Method overloading is a kind of polymorphism where different method implementations with the same name are invoked based on the types of the arguments passed. In mainstream languages, such as C++ and Java, overloading is typically a static mechanism in that the most appropriate implementation is selected statically by the compiler according to the static type of the arguments (*static overloading*). In this respect, Cardelli and Wegner [16] call this kind of polymorphism "ad hoc".

On the other hand, the need of adopting *dynamic overloading* (run-time selection mechanism based on the dynamic types of the receiver and the arguments) arises in many situations. A typical scenario is when there are two separate class hierarchies and the classes of a hierarchy have to operate on instances of classes of the other hierarchy according to their dynamic types. Quoting from [2], "*You have an operation that manipulates multiple polymorphic objects through pointers or references to their base classes. You would like the behavior of that operation to vary with the dynamic type of more than one of those objects*". This is quite a recurrent situation in object-oriented design, since separation of responsibilities enhances class decoupling and thus re-usability.

Typically, we have a class hierarchy for a specific operation with a superclass, say, `Operation`, that is separate from the hierarchy of the elements with a superclass, say, `Elem`. This separation easily permits changing dynamically the kind of operation performed on elements by simply changing the object responsible for the operation (indeed, object composition

☆ This work has been partially supported by the MIUR project EOS DUE.

* Corresponding author.
*E-mail addresses:* bettini@dsi.unifi.it (L. Bettini), capecchi@di.unito.it (S. Capecchi), venneri@dsi.unifi.it (B. Venneri).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.01.007

is preferred to class inheritance, in situations where these changes have to take place dynamically, see [27]). For instance, we may structure the class `Operation` as follows, where `ElemB` is a subclass of `ElemA` and `ElemC` is a subclass of `ElemB`,

```
class Operation {
    public void op(ElemA e) { ... }
    public void op(ElemB e) { ... }
    public void op(ElemC e) { ... }
    ...
}
```

in order to perform a specific operation according to the actual type of the element. However, such methods like `op`, having different signatures, are considered overloaded, and thus, in languages such as C++ and Java, they are both type checked and selected statically: at compile time it is decided which version fits best the type declaration of the argument `e`. So, the following code:

```
Operation o = new Operation();
ElemA e = new ElemB();
...
o.op(e);
```

would not select dynamically the most appropriate method according to the actual (run-time) type of `e`.

Dynamic overloading is useful also because it provides safe *covariant specialization* of methods, where subclasses are allowed to redefine a method by specializing its arguments. There is a general evidence that covariant code specialization is an indispensable practice in many situations [36,6]. Its most expressive application appears with *binary methods* [14], i.e., methods that act on objects of the same type: the receiver and the argument. It seems quite natural to specialize binary methods in subclasses and to require that the new code replaces the old definition when performing code selection at run-time according to the actual type of the arguments and the receiver.

Covariant specialization, overriding, overloading and binary methods have often participated together in making the relation between overloading and subtyping more complex than it is due. It is generally recognized that static overloading is misleading in Java-like languages [37,10]. Indeed, overloading interaction with inheritance differs among different languages, thus originating a confusing situation in programming in practice: it might become difficult to figure out which method implementation will be executed at run-time when different signatures match with the type of the actual parameters (by subtyping). Instead dynamic overloading does not suffer from this problem. In this respect, Castagna [17] provides an insightful study of these problems and concludes that once the above concepts are given the correct interpretation, everything turns out to be clear and, most importantly, safe. However, overloading needs to be dynamic in order to have the full flexibility without problems. Also for this reason, we believe that problems and skepticism in using static overloading should disappear when one can exploit dynamic overloading.

The concept of dynamic overloading is interchangeable with the one of *multi-methods* [23,39,18]. A multi-method can be seen as a collection of overloaded methods, called *branches*, associated to the same message, but the selection takes place dynamically according to run-time types of both the receiver and the arguments. Though multi-methods are widely studied in the literature and supported in languages such as *CLOS* [12], *Dylan* [43] and *BeCecil* [21], they have not been added to mainstream programming languages such as Java, C++ and C#. Due to this shortcoming, programmers are forced to resort to RTTI (run-time type information) mechanisms and **if** statements to manually explore the run-time type of an object, and to type downcasts, in order to force the view of an object according to its run-time representation. Indeed, in many cases, this is the only solution to overcome the shortcomings of the language [33,32]. However, the solutions based on RTTI and type casts are usually unsatisfactory since they make the evolution of the program harder by undermining its extensibility.

Moreover, the problem of solutions based on run-time checks is that, apart from requiring manual programming, they are error prone; e.g., in the following code, that tries to manually simulate dynamic overloading, the case for `ElemC` should have come before the case for `ElemB`:

```
public void op(ElemA e) {
    if (e instanceof ElemB)
        op((ElemB)e);
    else if (e instanceof ElemC)
        op((ElemC)e);
    else {
        // code for case ElemA
    }
}
```

As is, the case for `ElemC` will never be executed.

Another problem is that, when combining two object-oriented mechanisms, crucial issues arise that lead the designers of the language to choose directions that simplify the implementation of the language itself at the cost of sacrificing its flexibility. This typically happens when combining static overloading and inheritance [10]. In this respect, Ancona et al. [4]

tackle this issue and show how the design choices of languages such as Java (earlier than version 1.4) are counter intuitive and too restrictive. In particular, in [4] the authors point out that semantics of overloading and inheritance is rather "clean" if it is interpreted through a *copy semantics of inheritance*, whereby all the inherited overloaded methods are intended to be directly copied into the subclass (apart for those explicitly redefined by the subclass itself). This avoids a too restrictive overloading resolution policy that leads to "strange" compilation errors or unexpected behaviors. From a foundational point of view, following Meyer [37], inheritance nicely fits a framework where classes are viewed as functions, a special case of sets; the basic idea for modeling inheritance is that a class is the "union" of its own definition and those of its parent(s) (following the model of [1]). The "flattening" of classes is also adopted in engineering of object-oriented systems in particular to study the impact of inheritance on object-oriented metrics [11].

In this paper we propose FMJ (Featherweight Multi Java), an extension of Featherweight Java (FJ) [29,40] with multi-methods, thus providing dynamic overloading. Our proposal guarantees that the dynamic selection of an overloaded method is as specific as possible, also with respect to the parameter types, by using the standard method lookup starting from the dynamic class of the receiver. The extension is conservative and type safe (both "message-not-understood" and "message-ambiguous" are statically ruled out) and can be used as the formal basis for an actual implementation of dynamic overloading in Java-like languages.

FMJ was first introduced in [9], where multi-methods are constrained by three consistency conditions (that we call "global well-formedness" of multi-methods), adapted from [18,19], which require all the classes in the program to be inspected in order to statically rule out all possible ambiguities at run-time. In the present paper, we relax the notion of well-formedness, namely:

- we statically rule out run-time ambiguities by inspecting only multi-method invocation expressions (as in the compilers of Java-like languages), instead of definitions as in [9];
- the typechecking of multi-method definitions exploits a local well-formedness, which does not require the whole class hierarchy.

This makes our present model closer to programming language implementations where modules can be type checked in isolation and then composed dynamically without typechecking the whole program again. In a sense, the solution presented in [9] limits the object-oriented principle of extensibility: adding new subclasses, and then new branches for a multi-method, is rather intrusive on the previous code since it can undermine the soundness of superclasses, thus complicating maintenance and incremental software development. These drawbacks seem to be intrinsic to multi-methods declared within classes, so that many proposals in the literature prefer to adopt them as external functions (see Section 6).

In the present paper we improve the solution presented in [9], by avoiding limitations on modularity, program extensibility and maintenance, while retaining the basic principle that "methods are class members". Moreover we fully formalize our proposal by proving the type safety property. Once a program passes the typechecking (well typed), the absence of dynamic ambiguities is assured by a precompilation process that annotates method invocation expressions with static type information (collected during type checking). This type information will drive dynamic method dispatch to select the most appropriate (i.e., specialized) branch at run-time.

The annotation process allows us to model static overloading in a smooth way. Being able to model also static overloading is important, not only for studying the above mentioned interactions with other language features, but also for formalizing implementations of dynamic overloading through static overloading, such as [7,8].

The multi-methods we are considering are *encapsulated* multi-methods, i.e., they are actual methods of classes and not external functions (as, e.g., in [43,41]), and the branch selection is *symmetric*: during dynamic overloading selection the receiver type of the method invocation has no precedence over the argument types (differently from the encapsulated multi-methods of [14] and [13]).

Symmetry of branch selection is tightly related to copy semantics: if the receiver of a method invocation had the precedence over the parameters, we would search for the best matching branch only in the class of the receiver (or in the first superclass that defines some branch in case the class of the receiver does not define branches), i.e., without inspecting also the superclasses' branches (which could provide a more specialized version). Thus, copy semantics would not be implemented.

For instance, consider a subclass of the class `Operation`, where we redefine the branch for `ElemB` and we add a specialization to the multi-method with the branch for `ElemD` (where `ElemD` is a subclass of `ElemC`)[1]:

```
class ExtendedOperation extends Operation {
    public void op(ElemB e) { ... } // redefinition
    public void op(ElemD e) { ... } // specialization
    ...
};
```

If we had no copy semantics, then the following method invocation

[1] Throughout the paper, in the examples, we will use the full Java syntax instead of the smaller FMJ one (e.g., we will use imperative features). These examples could be written also in FMJ but they would only be more verbose.

| | | | |
|---|---|---|---|
| L | ::= | class C extends C {$\overline{\texttt{C}}$ $\overline{\texttt{f}}$; K; $\overline{\texttt{M}}$} | classes |
| K | ::= | C($\overline{\texttt{C}}$ $\overline{\texttt{f}}$){super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;} | constructors |
| M | ::= | C m ($\overline{\texttt{C}}$ $\overline{\texttt{x}}$){return e;} | methods |
| e | ::= | x \| e.f \| e.m($\overline{\texttt{e}}$) \| new C($\overline{\texttt{e}}$) | expressions |
| v | ::= | new C($\overline{\texttt{v}}$) | values |

**Fig. 1.** FMJ syntax.

```
Operation o = new ExtendedOperation();
ElemA e = new ElemC();
...
o.op(e);
```

would select the branch for `ElemB` in `ExtendedOperation`, while we should have selected the branch for `ElemC` in `Operation` which would be more specific.

In our approach, we get the desired behavior by adopting copy semantics and then symmetric selection. This way, the programmer of the subclasses is not required to redefine all branches just to avoid hiding the branches of the superclasses (this would also force the programmer to be aware of too many details of the superclasses). Although copy semantics is a useful mechanism in most situations, there might still be cases where the programmer of the derived class would actually like to "shadow" the branches of the superclass with the ones in the derived class. We refer to [9], Section 4, for some variations of FMJ without copy semantics. Furthermore, the overloading resolution policy in Java has followed copy semantics since version 1.4.

Concluding, we believe that the present proposal is interesting since, to the best of our knowledge, this is the first attempt to extend FJ with static and dynamic overloading (although some proposals for multi-methods in Java are already present, see, e.g., [13,22]). Furthermore, it shows that we can rule out possible run-time ambiguities during static type checking in a "modular" way, without requiring to inspect all the classes of a program; this absence of modularity was a crucial limitation of the first version of FMJ inherited from [18,19]. Moreover, we maintain the type safety of the language, differently from other proposals that model and implement multi-methods without preserving this crucial property (see Section 6).

Finally, we believe that dynamic overloading gives object-oriented languages an extra bit of flexibility and enhances code reuse, by giving the programmer the ability to develop programs in a natural way, without needing to resort to manual (and unsafe) ad hoc implementations.

The paper is organized as follows: in Section 2 we present syntax, typing and operational semantics of FMJ. In Section 3 we prove the main formal properties. Section 4 presents a version of FMJ with static overloading, exploiting the same annotation mechanism of Section 2. Section 5 further discusses our main design choices. Section 6 resumes some work on multi-methods in the literature. Finally, Section 7 concludes the paper and proposes some future directions.

## 2. Featherweight Multi Java

In this section we present syntax, typing and operational semantics of our proposal, the core language FMJ (*Featherweight Multi Java*), which is an extension of FJ (*Featherweight Java*) with multi-methods. FJ [29,40] is a lightweight version of Java, which focuses on a few basic features: mutually recursive class definitions, inheritance, object creation, method invocation, method recursion through `this`, subtyping and field access.[2] Thus, the minimal syntax, typing and semantics make the type safety proof simple and compact, in such a way that FJ is a handy tool for studying the consequences of extensions and variations with respect to Java ("FJ's main application is modeling extensions of Java", [40], page 248). Although we assume that the reader is familiar with FJ, we will give a brief introduction to FJ and then we will focus on the novel aspects of FMJ w.r.t. FJ.

The abstract syntax of FMJ constructs is given in Fig. 1 and it is just the same as FJ. The programmer, however, is allowed to specify more than one method with the same name and different signatures (multi-methods), and this difference w.r.t. FJ will be evident in the typing and in the semantics. The metavariables B, C, D and E range over class names; f and g range over attribute names; x ranges over method parameter names; e and d range over expressions and v and u range over values.

We use the overline notation for possibly empty sequences (e.g., "$\overline{\texttt{e}}$" is a shorthand for a possibly empty sequence "$\texttt{e}_1, \ldots, \texttt{e}_n$"). The length of a sequence $\overline{\texttt{e}}$ is denoted by $|\overline{\texttt{e}}|$. We abbreviate pair of sequences in similar way, e.g., $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ stands for $\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$. The empty sequence is denoted by $\bullet$.

Following FJ, we assume that the set of variables includes the special variable `this` (implicitly bound in any method declaration), which cannot be used as the name of a method's formal parameter (this restriction is imposed by the typing rules). Note that since we treat `this` in method bodies as an ordinary variable, no special syntax for it is required.

[2] FJ also includes up and down casts; however, since these features are completely orthogonal to our extension with multi-methods, they are omitted in FMJ.

$$\texttt{C} <: \texttt{C} \qquad \frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}} \qquad \frac{\texttt{class C extends D}\,\{\ldots\}}{\texttt{C} <: \texttt{D}}$$

**Fig. 2.** FMJ subtyping.

A class declaration `class C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\}$ consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field names $\overline{\texttt{C}}\ \overline{\texttt{f}}$ with their types, the constructor `K`, and a list of method definitions $\overline{\texttt{M}}$. The instance variables of `C` are added to the ones declared by `D` and its superclasses and are assumed to have distinct names. The constructor declaration shows how to initialize all these fields with the received values. A method definition `M` specifies the name, the signature and the body of a method; a body is a single `return` statement since FJ is a functional core of Java. The same method name can occur with different signatures in $\overline{\texttt{M}}$. Differently, in FJ all method names in the same class are assumed to be distinct and if a method with the same name is declared in the superclass, then it must have the same signature in all subclasses. The novel feature of FMJ consists in interpreting any definition of a method `m` in a class `C` as the definition of a new branch of the multi-method `m`; the new branch is added to all the other branches of `m` that are inherited (if they are not redefined) from the superclasses of `C` (*copy semantics*); thus `m` is a multi-method that is associated to different branch definitions for different signatures. Clearly, the notion of standard method is subsumed by our definition as a multi-method with only one branch. In the following, we will write $\texttt{m} \in \overline{\texttt{M}}$ ($\texttt{m} \notin \overline{\texttt{M}}$) to mean that at least one branch definition (no branch definition) of `m` belongs to $\overline{\texttt{M}}$. Finally, values, denoted by `v` and `u`, are fully evaluated object creation terms `new` $\texttt{C}(\overline{\texttt{v}})$ (i.e., expressions that are passed to the constructor are values too) and will be the results of completely evaluated well-typed expressions.

A class table *CT* is a mapping from class names to class declarations. Then a program is a pair (*CT*, `e`) of a class table (containing all the class definitions of the program) and an expression `e` (the program's main entry point). The class `Object` has no members and its declaration does not appear in *CT*. We assume that *CT* satisfies some usual sanity conditions: (i) $CT(\texttt{C}) = \texttt{class C} \ldots$ for every $\texttt{C} \in dom(CT)$; (ii) for every class name `C` (except `Object`) appearing anywhere in *CT*, we have $\texttt{C} \in dom(CT)$; (iii) there are no cycles in the transitive closure of the `extends` relation. Thus, in the following, instead of writing $CT(\texttt{C}) = \texttt{class C} \ldots$ we will simply write `class C` ....

The subtyping relation $<:$ on classes (types) is induced by the standard subclass relation (Fig. 2): $\texttt{C} <: \texttt{D}$ holds if and only if either `class C extends D` or `class C extends` $\texttt{D}_1$ and $\texttt{D}_1 <: \texttt{D}$. Thus, a subtyping relation $<:$ is defined for any fixed class table *CT*.

In order to focus the attention on our extension of FJ, we will discuss in the remainder of this section only the parts of typing and semantics that we introduced or that we modified w.r.t. FJ.

### 2.1. Multi-types

The types of FMJ are the types of FJ extended with *multi-types*, representing types of multi-methods. A multi-type is a set of arrow types associated to the branches of a multi-method, and is of the shape:

$$\{\overline{\texttt{C}}_1 \to \texttt{C}_1, \ldots, \overline{\texttt{C}}_n \to \texttt{C}_n\}$$

We will write multi-types in a compact form, by extending the sequence notation to multi-types:

$$\{\overline{\overline{\texttt{C}} \to \texttt{C}}\}$$

We extend the sequence notation also to multi-method definitions:

$$\overline{\texttt{C m}\ (\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}$$

represents a sequence of method definitions, each with the same name `m` but with different signatures (and possibly different bodies).

$$\texttt{C}_1\ \texttt{m}\ (\overline{\texttt{C}}_1\ \overline{\texttt{x}})\{\texttt{return e}_1\texttt{;}\}\ \ldots\ \texttt{C}_n\ \texttt{m}\ (\overline{\texttt{C}}_n\ \overline{\texttt{x}})\{\texttt{return e}_n\texttt{;}\}$$

The multi-type of the above multi-method will be denoted by $\{\overline{\overline{\texttt{C}} \to \texttt{C}}\}$.

To lighten the notation, in the following, we will assume a fixed class table *CT* and then $<:$ is the subtype relation induced by *CT*. We will write $\overline{\texttt{C}} <: \overline{\texttt{D}}$ as a shorthand for $\texttt{C}_1 <: \texttt{D}_1 \wedge \cdots \wedge \texttt{C}_n <: \texttt{D}_n$.

Multi-types can be constrained by three crucial consistency conditions (adapted from [18,19]), which must be checked statically, presented in the following definition of well-formedness.

**Definition 2.1** (*Global Well-Formedness of Multi-Types*). A multi-type $\{\overline{\overline{\texttt{B}} \to \texttt{B}}\}$ is *well-formed*, denoted by

$$\mathit{well\text{-}formed}(\{\overline{\overline{\texttt{B}} \to \texttt{B}}\})$$

if $\forall\ (\overline{\texttt{B}}_i \to \texttt{B}_i), (\overline{\texttt{B}}_j \to \texttt{B}_j) \in \{\overline{\overline{\texttt{B}} \to \texttt{B}}\}$ the following conditions are verified:

1. $\overline{\texttt{B}}_i \neq \overline{\texttt{B}}_j$

2. $\overline{\mathtt{B}}_i <: \overline{\mathtt{B}}_j \Rightarrow \mathtt{B}_i <: \mathtt{B}_j$
3. for all $\overline{\mathtt{E}}$ maximal types in $LB(\overline{\mathtt{B}}_i, \overline{\mathtt{B}}_j)$ there exists $\overline{\mathtt{E}} \to \mathtt{E} \in \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$ for some $\mathtt{E}$

In the above definition, $LB()$ is the standard lower bound set, i.e., $LB(\overline{\mathtt{B}}_i, \overline{\mathtt{B}}_j) = \{\overline{\mathtt{C}} \mid \overline{\mathtt{C}} <: \overline{\mathtt{B}}_i \wedge \overline{\mathtt{C}} <: \overline{\mathtt{B}}_j\}$. The first condition is quite natural, in that it requires that all input types are distinct. The second condition guarantees that a branch specialization is safe: if statically a branch selection has a return type, and if dynamically a more specialized branch is selected, the return type is consistent with the static selection (it is a subtype). The third condition is crucial to rule out statically any possible ambiguities. Note that, if no ambiguity is present at compile time, no ambiguity is guaranteed to be present also at run-time.

The third condition of well-formedness is very powerful, since it permits detecting ambiguity situations when checking multi-method definitions, but also quite expensive and hard to implement in a real world application: it requires to have all the types involved in a program and to test them all (that is why we call this *global* in this context). Besides the overhead of the implementation of such condition (which, however, would not decrease the performance of the program itself, since it is computed only during compilation), it might not be practical for modular compilations (e.g., a library, where the user code is not known in advance) and it might not be checked in programs where classes are loaded dynamically (e.g., Java programs that rely on run-time class loading).

Indeed, the compilers of mainstream programming languages, such as C++ and Java, in the presence of overloaded methods only check that a method invocation does not raise an ambiguity. This means that the class declarations might contain multi-methods that are not well-formed w.r.t. the global class hierarchy (Definition 2.1): the compiler does not reject the program, provided the ambiguity does not show up in method invocation expressions. In this paper we adopt the same policy: ambiguities are checked at multi-method invocation time (by using only the supertypes involved in the multi-method invocation and definitions), not at multi-method definition time (by exploring all the possible subtypes of the parameters used in multi-method branches). Note that the first two conditions of well-formedness can still be checked when type checking class declarations, without requiring the whole program. Thus, we are now considering a relaxed definition of well-formedness, that we call *Local Well-formedness*:

**Definition 2.2** (*Local Well-Formedness of Multi-Types*). A multi-type $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$ is *well-formed*, denoted by

$$\mathit{well\text{-}formed}(\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\})$$

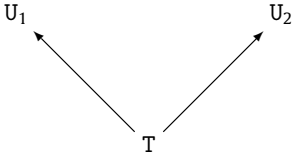
if $\forall\ (\overline{\mathtt{B}}_i \to \mathtt{B}_i), (\overline{\mathtt{B}}_j \to \mathtt{B}_j) \in \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$ the following conditions are verified:

1. $\overline{\mathtt{B}}_i \neq \overline{\mathtt{B}}_j$
2. $\overline{\mathtt{B}}_i <: \overline{\mathtt{B}}_j \Rightarrow \mathtt{B}_i <: \mathtt{B}_j$.

For static overloading languages, checking the absence of ambiguities at method invocation time is sufficient, since the method selection is performed according to static information on arguments and parameters. However, in our context of dynamic overloading, ruling out ambiguities upon method invocation time, relying on local well-formedness only, does not guarantee their absence at run-time, when types may decrease, unless we use some static type information during dynamic overloading method selection. We will describe our solution to this problem in the next sections, after a brief discussion about crucial cases of ambiguities in method invocations.

### 2.2. Multi-method selection: Dealing with ambiguities

Ambiguities are basically due to multiple inheritance on classes. In our context, as in FJ and in full Java, we do not deal with multiple inheritance; however ambiguities can still arise due to subtype relations on several (a sequence of) parameter types. For instance, let us consider the following type pairs: $\mathtt{T} = (\mathtt{B}', \mathtt{C}')$, $\mathtt{U}_1 = (\mathtt{B}', \mathtt{C})$ and $\mathtt{U}_2 = (\mathtt{B}, \mathtt{C}')$ where $\mathtt{B}' <: \mathtt{B}$ and $\mathtt{C}' <: \mathtt{C}$, then we obtain this subtyping diagram, which is similar to a multiple inheritance hierarchy:




Thus, for simplicity, when describing the example scenarios, we will consider multiple inheritance relations, since the same situations might be easily constructed with sequence of types easily (but they would be more verbose).

Let us consider the class hierarchy of Fig. 3 with three branches for the multi-method `m` that is used in the body of `n` (we are not considering return types and the class of the receiver). An invocation expression such as `v.m(new D(...))` would be rejected by the type system, since, statically, the method selection would be ambiguous; on the other hand, during static typing, `this.m(x)` is accepted, since `x` is of type $\mathtt{B}_1$, and it is typed by selecting the branch `m(`$\mathtt{B}_1$ `x)`. However, when we invoke `v.n(new D(...))`, then the above ambiguity arises only at run-time when `x` is replaced by the argument `new D(...)`.

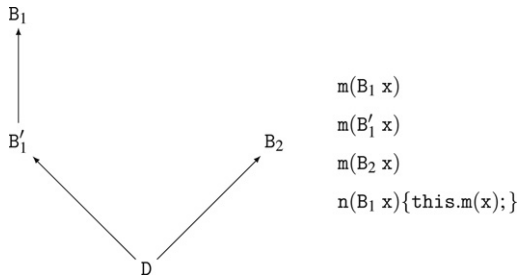



**Fig. 3.** A situation with possible ambiguities.

**Field lookup**

$$fields(\texttt{Object}) = \bullet$$

$$\frac{\texttt{class C extends D}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad fields(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

**Method type lookup**

$$mtype(\_, \texttt{Object}) = \emptyset$$

$$\frac{\texttt{class C extends D}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad \overline{\texttt{B m}\ (\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}} \in \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}) = \{\overline{\overline{\texttt{B}} \to \texttt{B}}\} \cup mtype(\texttt{m}, \texttt{D})}$$

$$\frac{\texttt{class C extends D}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}) = mtype(\texttt{m}, \texttt{D})}$$

$$\frac{mtype(\texttt{m}, \texttt{C}) = \{\overline{\overline{\texttt{B}} \to \texttt{B}}\} \qquad minsel(\overline{\texttt{C}}, \{\overline{\overline{\texttt{B}} \to \texttt{B}}\}) = \overline{\texttt{D}} \to \texttt{D}}{mtypesel(\texttt{m}, \texttt{C}, \overline{\texttt{C}}) = \overline{\texttt{D}} \to \texttt{D}}$$

**Fig. 4.** Lookup functions for typing.

We would like to stress that this situation should not represent a real ambiguity: within the context of the method n, x is considered of type $\texttt{B}_1$, and we only need to select a branch of the multi-method that handles arguments of type $\texttt{B}_1$ or, thanks to dynamic overloading, possible subtypes of $\texttt{B}_1$. Thus, such invocation can handle also values of type D at run-time, provided we "remember" that during static typing (i.e., during compilation) we had selected the branch $\texttt{m}(\texttt{B}_1\ \texttt{x})$. This means that we should completely rule out the branch for $\texttt{B}_2$, which is unrelated w.r.t. to this invocation static context. We observe that this will not prevent us to select the most specialized branch at run-time, i.e., $\texttt{m}(\texttt{B}'_1\ \texttt{x})$.

Summarizing, at run-time it is sound to select only a specialization of the static type. For instance, in the example above, the dynamic selection of $\texttt{m}(\texttt{B}_2\ \texttt{x})$ is unsound since its return type might be completely unrelated to the one of $\texttt{m}(\texttt{B}_1\ \texttt{x})$; in fact, condition 2 of well-formedness permits this, since $\texttt{B}_1$ and $\texttt{B}_2$ are unrelated and then the corresponding return types can be unrelated too.

Thus, if we *annotate* all method invocations with the type of the branch selected during static type checking, we can use this type information to filter the branches to inspect, when performing dynamic overloading method selection. Indeed, we will define a procedure to select the most appropriate branch at run-time using both the dynamic type of the arguments and the annotated static type, in such a way that no ambiguity can dynamically occur in well-typed programs. We will deal with technical details of this issue in the following sections.

### 2.3. Auxiliary functions

We define, in Fig. 4, some auxiliary functions to look up fields and method branches from *CT*; these functions are used in the typing rules and in the operational semantics.

The look up function *fields*(C) returns the sequence of the field names, together with the corresponding types, for all the fields declared in C and in its superclasses.

The *mtype*(m, C) lookup function (where m is the method name we are looking for, and C is the class where we are performing the lookup) differs from the one of FJ in that it does not select a method's signature, but it represents a multi-type. In particular, since we consider copy semantics of inheritance, the multi-type contains both the signatures of the branches defined (or redefined) in the current class and the ones inherited by the superclass. For any method name, *mtype* returns the empty set, when invoked on Object.

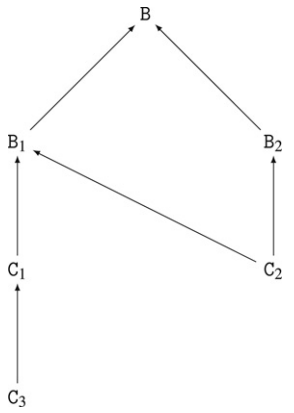



**Fig. 5.** A class hierarchy for lookup functions examples.

We introduce an additional lookup function, *mtypesel*($\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}$), that selects the signature of the branch of the multi-method $\mathtt{m}$ in the class $\mathtt{C}$, that matches best the specified argument types $\overline{\mathtt{C}}$ (function *minsel*). This consists in selecting, among the branches whose parameter types are supertypes of $\overline{\mathtt{C}}$, the one that is the most specific, avoiding ambiguities. The function *minsel* is defined formally in Definition 2.5 after some preliminary definitions.

**Definition 2.3** (*Minimal Parameter Types*). Given a set of arrow types $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$, we denote by $MIN(\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\})$ the set of *minimal* arrow types computed w.r.t. to the input types only, i.e.,

$$MIN(\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}) \stackrel{def}{=} \{\overline{\mathtt{B}}_i \to \mathtt{B}_i \in \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\} \mid \forall(\overline{\mathtt{B}}_j \to \mathtt{B}_j) \in \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\} \text{ s.t. } \overline{\mathtt{B}}_i \neq \overline{\mathtt{B}}_j, \overline{\mathtt{B}}_j \not<: \overline{\mathtt{B}}_i\}$$

For instance, let us consider the hierarchy in Fig. 5 and the multi-type $S = \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{B}_2 \to \mathtt{B}'_2, \mathtt{C}_1 \to \mathtt{C}'_1, \mathtt{C}_2 \to \mathtt{C}'_2\}$ then $MIN(S) = \{\mathtt{C}_1 \to \mathtt{C}'_1, \mathtt{C}_2 \to \mathtt{C}'_2\}$.

Note that *MIN*() is computed only on the parameter types, i.e., the return type is not considered (in fact, the return type does not participate in overloading selection). In particular, the following definitions, which rely on the subtyping relation on classes, inspect only the parameter types.

**Definition 2.4** (*Matching Parameter Types*). Given some parameter types $\overline{\mathtt{C}}$ and a multi-type $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$, we define:

$$match(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}) \stackrel{def}{=} \{\overline{\mathtt{B}}_j \to \mathtt{B}_j \in \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\} \mid \overline{\mathtt{C}} <: \overline{\mathtt{B}}_j\}$$

Thus, $match(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\})$ selects from the multi-type $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$ all the branches whose parameter types "match" (i.e., are supertypes of) the argument types $\overline{\mathtt{C}}$. For instance let us consider again the hierarchy in Fig. 5 and the multi-type $S$ then $match(\mathtt{B}_1, S) = \{\mathtt{B} \to \mathtt{B}'\}$ and $match(\mathtt{C}_3, S) = \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{C}_1 \to \mathtt{C}'_1\}$.

**Definition 2.5** (*Most Specialized Selection*). Given some parameter types $\overline{\mathtt{C}}$ and a multi-type $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$, then $minsel(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}) \stackrel{def}{=} \overline{\mathtt{B}}_i \to \mathtt{B}_i$ if and only if

$$MIN(match(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\})) = \{\overline{\mathtt{B}}_i \to \mathtt{B}_i\}$$

i.e., *MIN*() returns a singleton.

Thus, $minsel(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\})$ can fail either because the set of matching parameters is empty or because the $MIN(match(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}))$ returns a set that is not a singleton; in this latter case we have a static ambiguity, since more than one branch might be selected. For instance (we refer to Fig. 5):

- $minsel(\mathtt{C}_2, \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{B}_2 \to \mathtt{B}'_2\})$ is undefined
- $minsel(\mathtt{C}_2, \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{B}_2 \to \mathtt{B}'_2, \mathtt{C}_2 \to \mathtt{C}'_2\}) = \mathtt{C}_2 \to \mathtt{C}'_2$
- $minsel(\mathtt{B}_2, \{\mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{C}_1 \to \mathtt{C}'_1\})$ is undefined.

### 2.4. Typing

As usual, a type judgment of the form $\Gamma \vdash \mathtt{e} : \mathtt{C}$ states that "$\mathtt{e}$ has type $\mathtt{C}$ in the type environment $\Gamma$". A type environment is a finite mapping from variables (including `this`) to types, written $\overline{\mathtt{x}} : \overline{\mathtt{C}}$. Again, we use the sequence notation for abbreviating $\Gamma \vdash \mathtt{e}_1 : \mathtt{C}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{C}_n$ to $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$.

Typing rules are presented in Fig. 6. Concerning expressions, the only rule that changes w.r.t. FJ is T-Invk: method invocation concerns dynamic overloading, i.e., multi-method selection; in fact we use the *mtypesel* function, in order to obtain the return type of the selected branch.

**Expression typing**

$$\frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \text{T-VAR}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{C}_i}\ \text{T-FIELD}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) = \overline{\mathtt{B}} \to \mathtt{B}}{\Gamma \vdash \mathtt{e}.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{B}}\ \text{T-INVK}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}}\ \text{T-NEW}$$

**Method and Class typing**

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} : \mathtt{E} \qquad \mathtt{E} <: \mathtt{B}}{\mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}\ \mathtt{OK\ IN\ C}}\ \text{T-METHOD}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \{\overline{\mathtt{B} \to \mathtt{B}}\} \ \wedge\ \mathit{well\text{-}formed}(\{\overline{\mathtt{B} \to \mathtt{B}}\}) \text{ for all } \mathtt{m} \in \overline{\mathtt{M}}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\}\ \mathtt{OK}}\ \text{T-CLASS}$$

**Fig. 6.** Typing rules.

The rule for typing a branch, T-METHOD, differs from the original T-METHOD of FJ in that we do not check for signature redefinition (the *override* predicate of [40]). In FJ we need to check that if a subclass overrides a method then the signature is not modified, i.e., that we are overriding a method definition by preserving the signature. In our case, any new signature is legal: if it is the same as the signature of a branch already defined, then this redefinition overrides the implementation of that branch, otherwise it is intended as the definition of a new branch for that method.

Finally, T-CLASS checks also that any multi-method defined in the class is well-formed (the function *well-formed*, Definition 2.2).

### 2.5. Operational semantics

As discussed in Section 2.2, we need to annotate method invocation expressions with the (arrow) type of the branch selected during static type checking. Thus, the operational semantics is defined on *annotated programs*, i.e., FMJ programs where all expressions (including class method bodies) are annotated using the annotation function $\mathcal{A}$. Since this function relies on the static type system, it is parameterized over a type environment $\Gamma$.

**Definition 2.6** (*Annotation Function*). Given an expression $\mathtt{e}$ and a type environment $\Gamma$, the annotation of $\mathtt{e}$ w.r.t. $\Gamma$, denoted by $\mathcal{A}[\![\mathtt{e}]\!]_\Gamma$, is defined on the syntax of $\mathtt{e}$, by case analysis:

- $\mathcal{A}[\![\mathtt{x}]\!]_\Gamma = \mathtt{x}$
- $\mathcal{A}[\![\mathtt{e.f}]\!]_\Gamma = \mathcal{A}[\![\mathtt{e}]\!]_\Gamma.\mathtt{f}$
- $\mathcal{A}[\![\mathtt{e.m}(\overline{\mathtt{e}})]\!]_\Gamma = \mathcal{A}[\![\mathtt{e}]\!]_\Gamma.\mathtt{m}(\mathcal{A}[\![\overline{\mathtt{e}}]\!]_\Gamma)^{\overline{\mathtt{B}} \to \mathtt{B}}$
  if $\Gamma \vdash \mathtt{e} : \mathtt{C}$, $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$ and $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) = \overline{\mathtt{B}} \to \mathtt{B}$
- $\mathcal{A}[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_\Gamma = \mathtt{new\ C}(\mathcal{A}[\![\overline{\mathtt{e}}]\!]_\Gamma)$

Given a method definition $\mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\}$, in class $\mathtt{C}$, the annotation of the method body $\mathtt{e}$, is defined as $\mathcal{A}[\![\mathtt{e}]\!]_{\overline{\mathtt{x}}:\overline{\mathtt{B}},\mathtt{this}:\mathtt{C}}$.

In a real implementation, such annotation would be performed directly during compilation, i.e., during type checking. However, in the formal presentation, separating the two phases (type checking and annotation) makes the formal presentation simpler.

To further simplify the presentation of the operational semantics and of the properties, with an abuse of notation, in the following we will use $\mathtt{e}$ (and $\overline{\mathtt{e}}$) also for annotated expressions where not ambiguous.

**Method type lookup**

$$\frac{mtype(\mathtt{m},\mathtt{C}) = \{\overline{\mathtt{B} \to \mathtt{B}}\} \qquad bminsel(\overline{\mathtt{C}}, \{\overline{\mathtt{B} \to \mathtt{B}}\}, \overline{\mathtt{E}}) = \overline{\mathtt{D}} \to \mathtt{D}}{bmtypesel(\mathtt{m},\mathtt{C},\overline{\mathtt{C}},\overline{\mathtt{E}}) = \overline{\mathtt{D}} \to \mathtt{D}}$$

**Method body lookup**

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}\ \mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \mathtt{B\ m}\ (\overline{\mathtt{B}\ \mathtt{x}})\{\mathtt{return\ e};\} \in \overline{\mathtt{M}}}{mbody(\mathtt{m},\mathtt{C},\overline{\mathtt{B}}) = (\overline{\mathtt{x}},\mathtt{e})}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}\ \mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mbody(\mathtt{m},\mathtt{C},\overline{\mathtt{B}}) = mbody(\mathtt{m},\mathtt{D},\overline{\mathtt{B}})}$$

$$\frac{bmtypesel(\mathtt{m},\mathtt{C},\overline{\mathtt{C}},\overline{\mathtt{E}}) = \overline{\mathtt{D}} \to \mathtt{D} \qquad mbody(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = (\overline{\mathtt{x}},\mathtt{e})}{mbodysel(\mathtt{m},\mathtt{C},\overline{\mathtt{C}},\overline{\mathtt{E}}) = (\overline{\mathtt{x}},\mathtt{e})}$$

**Fig. 7.** Lookup functions for operational semantics.

Then, we present the additional auxiliary lookup functions for the operational semantics in Fig. 7; clearly they act on annotated expressions. The key function is *bmtypesel* which is similar to *mtypesel* of Fig. 4 but it has the additional fourth parameter that is used as a bound for the *bmatch* function. The following definitions will explain the role of *bmtypesel* and of this additional parameter.

**Definition 2.7** (*Bounded Matching Parameter Types*). Given some parameter types $\overline{\mathtt{C}}$ and $\overline{\mathtt{E}}$ and a multi-type $\{\overline{\mathtt{B} \to \mathtt{B}}\}$, we define:

$bmatch(\overline{\mathtt{C}}, \{\overline{\mathtt{B} \to \mathtt{B}}\}, \overline{\mathtt{E}}) \stackrel{def}{=} \{\overline{\mathtt{B}}_j \to \mathtt{B}_j \in \{\overline{\mathtt{B} \to \mathtt{B}}\} \mid \overline{\mathtt{C}} <: \overline{\mathtt{B}}_j <: \overline{\mathtt{E}}\}$

For instance let us consider again the hierarchy in Fig. 5 then:

- $bmatch(\mathtt{C}_1, \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{B}_2 \to \mathtt{B}'_2, \mathtt{C}_1 \to \mathtt{C}'_1, \mathtt{C}_2 \to \mathtt{C}'_2\}, \mathtt{B}) = \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{C}_1 \to \mathtt{C}'_1\}$
- $bmatch(\mathtt{C}_3, \{\mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{B}_2 \to \mathtt{B}'_2, \mathtt{C}_1 \to \mathtt{C}'_1\}, \mathtt{B}) = \{\mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{C}_1 \to \mathtt{C}'_1\}$

Thus, $bmatch(\overline{\mathtt{C}}, \{\overline{\mathtt{B} \to \mathtt{B}}\}, \overline{\mathtt{E}})$ selects from the multi-type $\{\overline{\mathtt{B} \to \mathtt{B}}\}$ all the branches whose parameter types "match" (i.e., are supertypes of) the argument types $\overline{\mathtt{C}}$ provided that, the parameter types are subtypes of $\overline{\mathtt{E}}$. This shows that $\overline{\mathtt{E}}$ acts as an upper bound to filter out branches that are unrelated with the parameter types that were used for the static typing (see the discussion in Section 2.2). Indeed, we call such $\overline{\mathtt{E}}$ the *top filter*.

**Definition 2.8** (*Upper Bounds*). Given the sets of arrow types $\{\overline{\mathtt{C} \to \mathtt{C}}\}$ and $\{\overline{\mathtt{B} \to \mathtt{B}}\}$, such that $\{\overline{\mathtt{C} \to \mathtt{C}}\} \subseteq \{\overline{\mathtt{B} \to \mathtt{B}}\}$, we define

$UB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\}) \stackrel{def}{=} \{\overline{\mathtt{B}}_j \to \mathtt{B}_j \in \{\overline{\mathtt{B} \to \mathtt{B}}\} \mid \forall \overline{\mathtt{C}}_i \to \mathtt{C}_i \in \{\overline{\mathtt{C} \to \mathtt{C}}\}, \overline{\mathtt{C}}_i <: \overline{\mathtt{B}}_j\}$

Basically, $UB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\})$ is just the standard notion of *upper bounds* applied to sets of arrow types, and computed considering only the parameter types. Indeed $UB(\{\mathtt{C}_1 \to \mathtt{C}'_1, \mathtt{C}_3 \to \mathtt{C}'_3\}, \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1, \mathtt{B}_2 \to \mathtt{B}'_2, \mathtt{C}_1 \to \mathtt{C}'_1, \mathtt{C}_2 \to \mathtt{C}'_2\}) = \{\mathtt{B} \to \mathtt{B}', \mathtt{B}_1 \to \mathtt{B}'_1\}$ (see Fig. 5).

**Definition 2.9** (*Least Upper Bound*). Given the sets of arrow types $\{\overline{\mathtt{C} \to \mathtt{C}}\}$ and $\{\overline{\mathtt{B} \to \mathtt{B}}\}$, such that $\{\overline{\mathtt{C} \to \mathtt{C}}\} \subseteq \{\overline{\mathtt{B} \to \mathtt{B}}\}$, we define the following recursive function:

$LUB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\}) \stackrel{def}{=}$
**let** $S = MIN(UB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\}))$ **in**
**if** $S \equiv \emptyset$ **then** *fail*
**else if** $S \equiv \{\overline{\mathtt{B}}_i \to \mathtt{B}_i\}$ **then** $\overline{\mathtt{B}}_i \to \mathtt{B}_i$
**else** $LUB(S, \{\overline{\mathtt{B} \to \mathtt{B}}\})$

The above recursive function $LUB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\})$ tries to find the element of $\{\overline{\mathtt{B} \to \mathtt{B}}\}$ which is the *least upper bound* of the subset $\{\overline{\mathtt{C} \to \mathtt{C}}\}$. The search succeeds when $MIN(UB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\}))$ returns a singleton; in case it returns a set, the function calls itself with such resulting set as the new first argument; the search fails if $MIN(UB(\{\overline{\mathtt{C} \to \mathtt{C}}\}, \{\overline{\mathtt{B} \to \mathtt{B}}\}))$ returns an empty set, i.e., if the set of upper bounds is empty. However, in a well-typed program, this function never fails at run-time (see Section 3). Furthermore, since there are no cycles in the class relation graph, it is trivial to verify that this function always terminates.

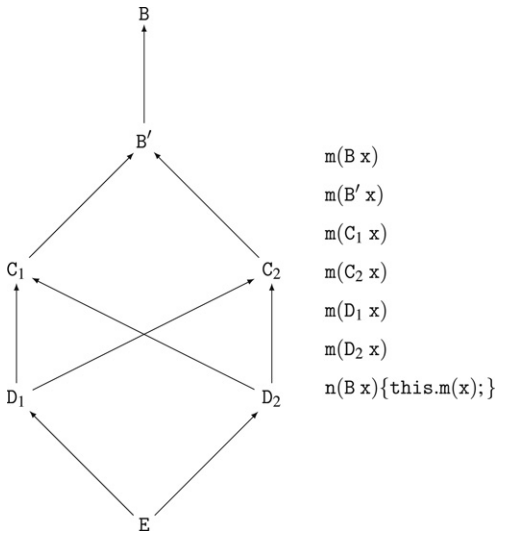



**Fig. 8.** A diamond hierarchy situation.

**Definition 2.10** (*Bounded Most Specialized Selection*). Given some parameter types $\overline{\mathtt{C}}$, $\overline{\mathtt{E}}$ and a multi-type $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}$, then $bminsel(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}, \overline{\mathtt{E}})$ is defined as follows

$$bminsel(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}, \overline{\mathtt{E}}) \stackrel{def}{=} LUB(MIN(bmatch(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}, \overline{\mathtt{E}})), bmatch(\overline{\mathtt{C}}, \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\}, \overline{\mathtt{E}}))$$

**Example 2.11.** Let us consider the crucial situation, with a diamond hierarchy, depicted in Fig. 8. The method definition $\mathtt{n(B\ x)\{this.m(x);\}}$ is well typed, and $\mathtt{this.m(x)}$ is annotated with the static type $\mathtt{B} \to \mathtt{B}_1$ ($\mathtt{B}_1$ is the return type of $\mathtt{m}$). If we perform the method invocation $\mathtt{v.n(new\ E(\ldots))}$, the B annotation will not prevent us from selecting both the branches $\mathtt{m(D_1\ x)}$ and $\mathtt{m(D_2\ x)}$ as possible minimal candidates. In particular (for simplicity, we are not considering return types):

$bmatch(\mathtt{E}, \{\mathtt{B}, \mathtt{B}', \mathtt{C}_1, \mathtt{C}_2, \mathtt{D}_1, \mathtt{D}_2\}, \mathtt{B}) = \{\mathtt{B}, \mathtt{B}', \mathtt{C}_1, \mathtt{C}_2, \mathtt{D}_1, \mathtt{D}_2\}$
$MIN(bmatch(\mathtt{E}, \{\mathtt{B}, \mathtt{B}', \mathtt{C}_1, \mathtt{C}_2, \mathtt{D}_1, \mathtt{D}_2\}, \mathtt{B})) = \{\mathtt{D}_1, \mathtt{D}_2\}$
Let us denote $bmatch(\mathtt{E}, \{\mathtt{B}, \mathtt{B}', \mathtt{C}_1, \mathtt{C}_2, \mathtt{D}_1, \mathtt{D}_2\}, \mathtt{B})$ by $M$.
Now, we invoke $LUB(\{\mathtt{D}_1, \mathtt{D}_2\}, M)$. Since
$MIN(UB(\{\mathtt{D}_1, \mathtt{D}_2\}, M)) = MIN(\{\mathtt{B}, \mathtt{B}', \mathtt{C}_1, \mathtt{C}_2\}) = \{\mathtt{C}_1, \mathtt{C}_2\}$,
which is not a singleton, then we perform the recursive call $LUB(\{\mathtt{C}_1, \mathtt{C}_2\}, M)$ which returns $MIN(UB(\{\mathtt{C}_1, \mathtt{C}_2\}, M))$ that is the singleton $\{\mathtt{B}'\}$. Thus we select the branch $\mathtt{m(B'\ x)}$. Note that, since we want to avoid run-time ambiguities while choosing the most specialized branch, then selecting the branch $\mathtt{m(B'\ x)}$ is the only safe choice.

Once the signature of the branch to select is determined, $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}})$ selects the body of the branch of the multi-method $\mathtt{m}$ in the class $\mathtt{C}$ that has the parameter types $\overline{\mathtt{C}}$ (since branches are inherited, this function may have to inspect the superclass of $\mathtt{C}$ in case that body is not defined in $\mathtt{C}$). *mbody* returns a pair $(\overline{\mathtt{x}}, \mathtt{e})$ where the first element is the parameter sequence, and the second one is the actual body of the selected branch. Differently from FJ, where the parameter types were not needed, since only one signature for a method can exist in a class hierarchy, here we also need to specify the parameter types in order to perform the body selection. We introduce the additional lookup function, $mbodysel(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{E}})$ that searches for the signature of the branch that matches best the argument types $\overline{\mathtt{C}}$ with $\overline{\mathtt{E}}$ as top filter, by using the above mentioned *mtypesel* lookup function, and then selects the body of the obtained signature, by using *mbody*.

The operational semantics (Fig. 9) is defined by the reduction relation $\mathtt{e} \longrightarrow \mathtt{e}'$, read "e reduces to e′ in one step". The standard reflexive and transitive closure of $\longrightarrow$, denoted by $\longrightarrow^\star$, defines the reduction relation in many steps. We adopt a deterministic call-by-value semantics, analogous to the call-by-value strategy of FJ presented in [40]. The congruence rules define that the operators (method invocation, object creation and field selection) are reduced only when all their subexpressions are reduced to values (call-by-value).

The expression $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \mathtt{u}]\mathtt{e}$ denotes the expression obtained from e by replacing $\mathtt{x}_1$ with $\mathtt{v}_1, \ldots, \mathtt{x}_n$ with $\mathtt{v}_n$ and this with u.

We focus on the only reduction rule that needs to be changed w.r.t. FJ, i.e., the method selection rule. The rule R-Invk employs dynamic overloading for selecting the most specific body (branch) of the invoked method, with respect to the actual types of both the receiver and the arguments. The semantics of method invocation is type driven: the function *bmtypesel* is employed by *mbodysel* to obtain the signature of the most specialized branch, then the function *mbody* searches for the body that is associated to this signature starting from the actual type of the receiver object (Fig. 4). Thus, the selected

**Reduction**

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new\ C}(\overline{\mathtt{v}})).\mathtt{f}_i \longrightarrow \mathtt{v}_i}\ \text{R-FIELD}$$

$$\frac{\mathit{mbodysel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{D}}, \overline{\mathtt{E}}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{new\ D}(\overline{\mathtt{u}})})^{\overline{\mathtt{E}}\to\mathtt{E}} \longrightarrow [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{new\ D}(\overline{\mathtt{u}})}, \mathtt{this} \leftarrow \mathtt{new\ C}(\overline{\mathtt{v}})]\mathtt{e}_0}\ \text{R-INVK}$$

**Congruence rules**

$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{e.f} \longrightarrow \mathtt{e}'.\mathtt{f}} \qquad \frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{e.m}(\overline{\mathtt{e}})^{\overline{\mathtt{C}}\to\mathtt{C}} \longrightarrow \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}})^{\overline{\mathtt{C}}\to\mathtt{C}}}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}})^{\overline{\mathtt{C}}\to\mathtt{C}} \longrightarrow \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{e}_i', \overline{\mathtt{e}})^{\overline{\mathtt{C}}\to\mathtt{C}}} \qquad \frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C}(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{v}}, \mathtt{e}_i', \overline{\mathtt{e}})}$$

**Fig. 9.** Operational semantics.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{D}} \to \mathtt{D} \in \mathit{mtype}(\mathtt{m}, \mathtt{C}) \qquad \mathit{bmtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{D}}) = \overline{\mathtt{B}} \to \mathtt{B}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}})^{\overline{\mathtt{D}}\to\mathtt{D}} : \mathtt{D}}\ \text{TA-INVK}$$

**Fig. 10.** Typing rule for annotated method invocation ($\mathtt{e}$ and $\overline{\mathtt{e}}$ are intended to denote annotated expressions).

method body is not only the most specialized w.r.t. the argument types but also the most redefined version associated to that signature. This way, we model standard method overriding inside our mechanism of dynamic overloading. Note that we pass to *mbodysel* the static annotation $\overline{\mathtt{E}}$, that will then be used as the top filter by *mtypesel*. This will guarantee the absence of dynamic ambiguities.

## 3. Properties

In this section we address the issue of type safety for FMJ.

Since our small-step semantics is defined on annotated expressions, we must use a type system that checks annotated expressions in order to formulate the type preservation theorem. Indeed, in compiled well-typed programs, method invocations are the only expressions that are type annotated. Then the type system we need in this section, for proving type safety, is just the same as the one in Fig. 6, but for the rule T-INVK which is replaced by the new rule TA-INVK. TA-INVK is presented in Fig. 10, where $\mathtt{e}$ and $\overline{\mathtt{e}}$ are intended to denote annotated expressions. It checks that (i) both $\mathtt{e}$ and $\overline{\mathtt{e}}$ are well typed (as in the rule T-INVK); (ii) the type annotation is correct since the annotated type belongs to the multi-type of the method that is invoked; (iii) the function *bmtypesel* is defined, that is the signature of the most specialized branch is univocally determined and is a subtype of the annotation, using the annotated type as a top filter. If the above conditions are true, then the invocation expression is given type $\mathtt{D}$, since the method is annotated with the type $\overline{\mathtt{D}} \to \mathtt{D}$. We will use the same derivation symbol $\vdash$ also for this type system; this makes sense thanks to Property 3.3.

**Property 3.1.** *Given a method name* $\mathtt{m}$*, a class name* $\mathtt{C}$*, and some parameter types* $\overline{\mathtt{C}}$*, if* $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) = \overline{\mathtt{B}} \to \mathtt{B}$ *then* $\mathit{bmtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{B}}) = \overline{\mathtt{B}} \to \mathtt{B}$.

**Proof.** Suppose that $\mathit{bmtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{B}}) = \overline{\mathtt{D}} \to \mathtt{D}$ with $\overline{\mathtt{D}} \neq \overline{\mathtt{B}}$; by Definition 2.7 we must have $\overline{\mathtt{D}} <: \overline{\mathtt{B}}$, but then, by Definition 2.5, $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) \neq \overline{\mathtt{B}} \to \mathtt{B}$ which is an absurdum. □

**Property 3.2.** *If* $\mathit{bmtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{D}}) = \overline{\mathtt{B}} \to \mathtt{B}$ *then*

(1) $\overline{\mathtt{B}} \to \mathtt{B} \in \mathit{mtype}(\mathtt{m}, \mathtt{C})$
(2) $\overline{\mathtt{B}} <: \overline{\mathtt{D}}$

**Proof.** The result follows from the definition of *bmtypesel* (Fig. 7). □

**Property 3.3** (*Types are Preserved Under Annotation*). *If* $\Gamma \vdash \mathtt{e} : \mathtt{B}$ *for some* $\Gamma$ *and* $\mathtt{B}$*, then* $\Gamma \vdash \mathcal{A}[\![\mathtt{e}]\!]_\Gamma : \mathtt{B}$.

**Proof.** By induction on a derivation of $\Gamma \vdash \mathtt{e} : \mathtt{B}$. The only interesting case is T-INVK, $\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{B}$; by rule T-INVK (Fig. 6) we have $\Gamma \vdash \mathtt{e} : \mathtt{C}$, $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$ and $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) = \overline{\mathtt{B}} \to \mathtt{B}$; by Definition 2.6, $\mathcal{A}[\![\mathtt{e.m}(\overline{\mathtt{e}})]\!]_\Gamma = \mathcal{A}[\![\mathtt{e}]\!]_\Gamma.\mathtt{m}(\mathcal{A}[\![\overline{\mathtt{e}}]\!]_\Gamma)^{\overline{\mathtt{B}}\to\mathtt{B}}$. By induction hypothesis $\Gamma \vdash \mathtt{e} : \mathtt{C}$ implies $\Gamma \vdash \mathcal{A}[\![\mathtt{e}]\!]_\Gamma : \mathtt{C}$ and $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$ implies $\Gamma \vdash \mathcal{A}[\![\overline{\mathtt{e}}]\!]_\Gamma : \overline{\mathtt{C}}$ and by Property 3.1 $\mathit{bmtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{B}}) = \overline{\mathtt{B}} \to \mathtt{B}$, where $\overline{\mathtt{B}} \to \mathtt{B} \in \mathit{mtype}(\mathtt{m}, \mathtt{C})$ by Property 3.2. Thus we can apply rule TA-INVK (Fig. 10) to obtain $\Gamma \vdash \mathcal{A}[\![\mathtt{e.m}(\overline{\mathtt{e}})]\!]_\Gamma : \mathtt{B}$. □

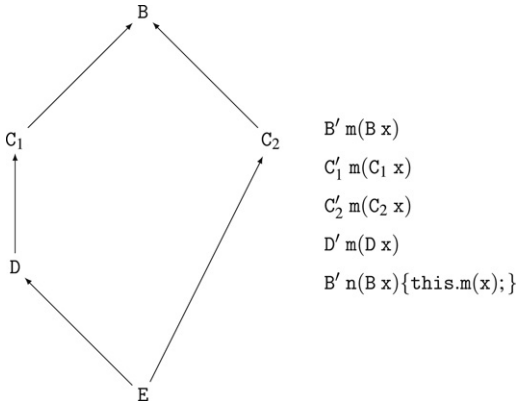



**Fig. 11.** A class hierarchy.

By Property 3.3 any well-typed program is compiled to an annotated program such that annotated expressions preserve their types, before its execution. Then we have to prove type preservation and progress properties for annotated programs, in order to guarantee the type safety of our language. In the following we will prove these properties after some preliminary Lemmas. By abusing notation we continue to use $\mathtt{e}$ and $\overline{\mathtt{e}}$ to denote annotated expressions; moreover, we assume that the programs are statically typed and then the class table is well typed.

**Lemma 3.4.** *If* $bmtypesel(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{D}}) = \overline{\mathtt{B}} \to \mathtt{B}$, *where* $\overline{\mathtt{D}} \to \mathtt{D} \in mtype(\mathtt{m}, \mathtt{C})$ *for some* D, *then, for any* $\mathtt{C}_1$ *and* $\overline{\mathtt{C}}_1$ *such that* $\mathtt{C}_1 <: \mathtt{C}$ *and* $\overline{\mathtt{C}}_1 <: \overline{\mathtt{C}}$, *we have that*
$bmtypesel(\mathtt{m}, \mathtt{C}_1, \overline{\mathtt{C}}_1, \overline{\mathtt{D}}) = \overline{\mathtt{E}} \to \mathtt{E}$ *where* $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$ *and* $\overline{\mathtt{D}} \to \mathtt{D} \in mtype(\mathtt{m}, \mathtt{C}_1)$.

**Proof.** Let $S$ and $S_1$ denote the following sets:
$S = bmatch(\overline{\mathtt{C}}, mtype(\mathtt{m}, \mathtt{C}), \overline{\mathtt{D}})$
$S_1 = bmatch(\overline{\mathtt{C}}_1, mtype(\mathtt{m}, \mathtt{C}_1), \overline{\mathtt{D}})$
Since $\mathtt{C}_1 <: \mathtt{C}$, we have
(1) $mtype(\mathtt{m}, \mathtt{C}_1) \supseteq mtype(\mathtt{m}, \mathtt{C})$
and then, since $\overline{\mathtt{C}}_1 <: \mathtt{C}$,
(2) $S_1 \supseteq S$
Since by assumption $bmtypesel(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}, \overline{\mathtt{D}})$ is defined, then we obtain that $S$ is not empty, which implies
(3) $S_1$ is not empty (from (2)).
Moreover, $\overline{\mathtt{D}} \to \mathtt{D} \in S$, since $\overline{\mathtt{D}} \to \mathtt{D} \in mtype(\mathtt{m}, \mathtt{C})$ and $\overline{\mathtt{C}} <: \overline{\mathtt{B}} <: \overline{\mathtt{D}}$; thus we have
(4) $\overline{\mathtt{D}} \to \mathtt{D} \in S_1$ (from (1) and (2))
Therefore $\overline{\mathtt{D}} \to \mathtt{D}$ is the least upper bound of $S_1$ (from (4) and by definition of *bmatch*) which implies $bmtypesel(\mathtt{m}, \mathtt{C}_1, \overline{\mathtt{C}}_1, \overline{\mathtt{D}}) = \overline{\mathtt{E}} \to \mathtt{E}$, for some E and $\overline{\mathtt{E}}$ with $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$. Finally, $\overline{\mathtt{D}} \to \mathtt{D} \in mtype(\mathtt{m}, \mathtt{C}_1)$ follows from the assumption $\overline{\mathtt{D}} \to \mathtt{D} \in mtype(\mathtt{m}, \mathtt{C})$ using (1). □

The previous Lemma is a crucial property to prove the Substitution Property. Note that if m is a standard method, whose redefinition in subclasses preserves the signature (standard method overriding), then, our definitions of *bmtypesel* and the proof of the following Substitution Lemma collapses to the proof of the analogous Lemma for FJ. Indeed, our language extension is conservative.

Let us observe that we cannot prove the more general form of the Lemma where $bmtypesel(\mathtt{m}, \mathtt{C}_1, \overline{\mathtt{C}}_1, \overline{\mathtt{D}}) = \overline{\mathtt{E}} \to \mathtt{E}$ where $\overline{\mathtt{E}} <: \overline{\mathtt{B}}$ (see Lemma 3.1 in [9]); this is because, during reduction of subexpressions in a method invocation expression, when decreasing the type of the receiver and the types of the arguments, the type of the branch used to type check the invocation expression might not decrease (but the parameter types of the selected branch will never be a supertype of the top filter $\overline{\mathtt{D}}$, as stated in Lemma 3.4).

For instance, let us consider the example in Fig. 11. The parameter of method n has static type B. At run-time this type can decrease to D. In this case the selected branch for the call $\mathtt{this.m(x)}$ is $\mathtt{m(D\ x)}$ since $bminsel(\mathtt{D}, \mathtt{M}, \mathtt{B}) = \mathtt{D} \to \mathtt{D}'$ where M is the set of arrow types whose input types match with D w.r.t the static filter B, $\mathtt{D}' <: \mathtt{B}'$ and $\mathtt{B}'$ is the return type of the branch $\mathtt{m(B\ x)}$. Then if the type decreases from D to E we have $bminsel(\mathtt{E}, \mathtt{M}', \mathtt{B}) = \mathtt{B} \to \mathtt{B}'$ (where $\mathtt{M}'$ is the set of arrow types whose input types match with E). Thus, the return type of the selected branch increased from $\mathtt{D}'$ to $\mathtt{B}'$.

This is due to the fact that going down in the class hierarchy there can be more superclasses (as in the case of E), thus, the set of matching branches is larger ($\mathtt{M} \subset \mathtt{M}'$) and we may have to go up in the hierarchy to find the least upper bound to solve possible diamond situations during method selection.

The type can actually decrease (w.r.t. the static type assigned to the original method invocation expression) only when all subexpressions are reduced to values and the method selection actually takes place as the next step in the reduction. This latter fact is the one that matters most, but in our one-step reduction strategy, we need to take into consideration also the above issue, otherwise we would not be able to prove the substitution lemma and the type preservation theorem. Thus,

in TA-Invk, the type assigned to the expression is always the static one, i.e., D. Then, Lemma 3.4 is enough to prove the substitution lemma and the type preservation theorem.

**Lemma 3.5** (*Substitution Lemma*). *If* $\Gamma, \overline{x} : \overline{B} \vdash e : D$ *and* $\Gamma \vdash \overline{e} : \overline{E}$ *where* $\overline{E} <: \overline{B}$, *then* $\Gamma \vdash [\overline{x} \leftarrow \overline{e}]e : C$ *for some* $C <: D$.

**Proof.** By induction on the derivation of $\Gamma, \overline{x} : \overline{B} \vdash e : D$, with a case analysis on the last applied rule.

T-Var. If $x \notin \overline{x}$ then the conclusion is immediate since $[\overline{x} \leftarrow \overline{e}]x = x$; otherwise, $x = x_i$ and $D = B_i$, and $[\overline{x} \leftarrow \overline{e}]x = [\overline{x} \leftarrow \overline{e}]x_i = e_i$; then, by assumption, $\Gamma \vdash e_i : E_i <: B_i$.

T-Field. By induction hypothesis, taking into account that fields, with their types, are inherited in subclasses and all field names are assumed to be distinct.

TA-Invk. $e = e_0.m(\overline{e}_0)^{\overline{D} \to D}$ and $\Gamma, \overline{x} : \overline{B} \vdash e_0.m(\overline{e}_0)^{\overline{D} \to D} : D$ for some $\overline{D}$ and D. The premises of TA-Invk are

- $\Gamma, \overline{x} : \overline{B} \vdash e_0 : C_1$,
- $\Gamma, \overline{x} : \overline{B} \vdash \overline{e}_0 : \overline{C}_1$,
- $bmtypesel(m, C_1, \overline{C}_1, \overline{D}) = \overline{D}_1 \to D_1$ where $\overline{D} \to D \in mtype(m, C_1)$.

Then, by induction hypothesis

- $\Gamma \vdash [\overline{x} \leftarrow \overline{e}]e_0 : C_2 <: C_1$, (1)
- $\Gamma \vdash [\overline{x} \leftarrow \overline{e}]\overline{e}_0 : \overline{C}_2 <: \overline{C}_1$. (2)

By Lemma 3.4, $bmtypesel(m, C_2, \overline{C}_2, \overline{D}) = \overline{D}_2 \to D_2$ where $\overline{D} \to D \in mtype(m, C_2)$ (3). Thus, using TA-Invk on (1), (2) and (3) we obtain

$\Gamma \vdash ([\overline{x} \leftarrow \overline{e}]e_0).m([\overline{x} \leftarrow \overline{e}]\overline{e}_0)^{\overline{D} \to D} : D$.

T-New. By induction hypothesis. □

**Theorem 3.6** (*Type Preservation*). *If* $\Gamma \vdash e : E$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : E'$ *for some* $E' <: E$.

**Proof.** By induction on a derivation of $e \longrightarrow e'$. The only crucial case is when the last applied rule is R-Invk:

$$\frac{mbodysel(m, C, \overline{D}, \overline{E}) = (\overline{x}, e_0)}{new\ C(\overline{v}).m(\overline{new\ D(\overline{u})})^{\overline{E} \to E} \longrightarrow [\overline{x} \leftarrow \overline{new\ D(\overline{u})}, this \leftarrow new\ C(\overline{v})]e_0}\ \text{R-Invk}$$

By assumption, $\Gamma \vdash new\ C(\overline{v}).m(\overline{new\ D(\overline{u})})^{\overline{E} \to E} : E$ which can be obtained only by TA-Invk where the premises are

(1) $\Gamma \vdash \overline{new\ C(\overline{v})} : C$.
(2) $\Gamma \vdash \overline{new\ D(\overline{u})} : \overline{D}$.
(3) $bmtypesel(m, C, \overline{D}, \overline{E}) = \overline{B} \to B$, where $\overline{E} \to E \in mtype(m, C)$, so that $\overline{B} <: \overline{E}$ (Property 3.2), and $\overline{B}$ matches $\overline{D}$, i.e., $\overline{D} <: \overline{B}$

Moreover, by definitions of auxiliary functions in Fig. 7, using (3), we have that $mbodysel(m, C, \overline{D}, \overline{E}) = (\overline{x}, e_0)$ implies $\overline{B} \to B \in mtype(m, C)$ and there is a superclass $C_1$ of C ($C <: C_1$) such that $this : C_1, \overline{x} : \overline{B} \vdash e_0 : B' <: B$ for some $B'$. Then, by using (1) and (2) and the Substitution Lemma 3.5, we have $\vdash [\overline{x} \leftarrow \overline{new\ D(\overline{u})}, this \leftarrow new\ C(\overline{v})]e_0 : B'' <: B'$. Finally, $\overline{B} <: \overline{E}$ implies $B <: E$, by well-formedness of multi-types (Definition 2.2-(2)) and then $B'' <: B' <: B <: E$.

The cases of congruence rules are straightforward, using Lemma 3.4 for method invocation expressions. □

**Theorem 3.7** (*Progress*). *Let* e *be a closed expression. If* $\vdash e : D$ *for some* D, *then either* e *is a value or* $e \longrightarrow e'$ *for some* $e'$.

**Proof.** By induction on the derivation of $\vdash e : B$, with a case analysis on the last applied rule.

T-Var is excluded since e is closed.
T-Field, T-New follow from the induction hypothesis, using congruence rules.
TA-Invk is the only crucial case, that is:

$$\frac{\Gamma \vdash e_0 : C \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad bmtypesel(m, C, \overline{C}, \overline{D}) = \overline{B} \to B \qquad \overline{D} \to D \in mtype(m, C)}{\Gamma \vdash e_0.m(\overline{e})^{\overline{D} \to D} : D}$$

By induction hypothesis, $e_0$ or $\overline{e}$ can be reduced or they are all values. In the first case the method invocation expression reduces by using congruence rules. Then, let us consider the particular case when all subexpressions are values, that is $e = new\ C(\overline{v}).m(\overline{new\ C(\overline{u})})^{\overline{D} \to D}$.

By Property 3.2 we have that $bmtypesel(m, C, \overline{C}, \overline{D}) = \overline{B} \to B$ implies $\overline{B} \to B \in mtype(m, C)$. By definition of *mtype* and since the class table *CT* is well typed, we have $mbody(m, C, \overline{D}) = (\overline{x}, e_1)$ for some $(\overline{x}, e_1)$. By definition of *mbodysel* in Fig. 7 we have $mbodysel(m, C, \overline{C}, \overline{D}) = (\overline{x}, e_1)$ thus we can use the rule R-Invk to obtain $[\overline{x} \leftarrow \overline{new\ C(\overline{u})}, this \leftarrow new\ C(\overline{v})]e_1$. □

Theorems 3.6 and 3.7 show how type safety of FJ can be preserved when adding multi-methods with dynamic overloading with our approach, i.e., any well-typed FMJ program cannot get stuck.

## 4. Extension to static overloading

Our approach can be used to model also *static overloading* in a smooth way. Indeed, we can extend the language in order to provide both static and dynamic overloading mechanisms, by adding the new expression $\texttt{st e.m}(\overline{\texttt{e}})$ to represent a method call in which the selection of the body must be performed according to the static overloading policy.

It is intuitively fairly clear that no specific new rules are needed as far as typing and congruence rules are concerned. We can use T-Invk also for (statically) type checking the new construct. Analogously, the operational congruence rules for method invocation can be extended to $\texttt{st e.m}(\overline{\texttt{e}})$, so preserving the call-by-value reduction strategy.

Instead, we have to add a specific reduction rule:

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}, \overline{\texttt{E}}) = (\overline{\texttt{x}}, \texttt{e}_0)}{\texttt{st new C}(\overline{\texttt{v}})\texttt{.m}(\overline{\texttt{new D}(\overline{\texttt{u}})})^{\overline{\texttt{E}} \to \texttt{E}} \longrightarrow [\overline{\texttt{x}} \leftarrow \overline{\texttt{new D}(\overline{\texttt{u}})}, \texttt{this} \leftarrow \texttt{new C}(\overline{\texttt{v}})]\texttt{e}_0} \text{ R-SInvk}$$

Since we have the (static) type annotation, we already have all the information we need to select the right branch: we simply use the argument types of the annotation to select the body by means of the function *mbody*. This highlights the fact that method selection is performed once and for all at compile time. Note however, that once the signature $\overline{\texttt{E}} \to \texttt{E}$ has been selected via static overloading resolution, dynamic binding is still employed to select at run-time the most specialized body of m according to the dynamic type of the receiver.

In our approach, "the compile time selection" is implemented through the annotations which "remember" the types that are used during type checking to statically select the branch. In programming language implementations, a technique to perform the annotation is *name mangling*, as in C++ [46,35]: a way of encoding additional information in the name of a function in order to pass more information from the compilers to linkers. Thus, the C++ compiler adds information about the parameters of a function (or method) to the name of the function itself. Our annotation can be seen as a formalization of the mentioned name mangling technique.

We stressed, in the Introduction, that static overloading is generally considered a misleading mechanism because of its complicated interaction with inheritance. Then, why should we treat it? Firstly, overloading in Java (as in most mainstream programming languages) is static and this feature has not been modeled yet in Featherweight Java which is widely used to model Java extensions. Then having static overloading in the FJ model permits isolating and studying the contexts where overloading interacts badly with inheritance; furthermore, it permits investigating possible interactions with other features of a proposed language extension. Secondly, the presence of static overloading in a core language can be used to study the simulation of dynamic overloading by using static overloading and dynamic binding, following the idea of Ingalls [30]. Thus, we can use our extension to formalize the transformation from an extended Java with dynamic overloading to standard Java. The formalization will prove not only that the translated program is still type safe, but also that the semantics of the translated program is the same as the semantics of the original one, providing a framework to reason about correctness of compilers when adding dynamic overloading to a language (see also Section 7).

We conclude this section by observing that the type annotation technique would be useful also for handling super: as suggested in [40] (page 531) in order to handle super, "we need some way of remembering what class the currently executing method came from". Again, such information can be gathered during typing and recorded in an annotation, before executing the program.

## 5. Discussion on FMJ design choices

In this section we point out and motivate main design choices in the type system and the operational semantics of FMJ.

The annotation technique is a key point in our approach, since it permits avoiding dynamic ambiguity situations, like the one depicted in Fig. 3. The need to introduce this technique is mainly due to the fact that two different lookup functions are required for selecting the most specialized branch of a method invocation, one in the static typing and the other at run-time. The type system detects static ambiguity situations by using the lookup function *minsel* (Definition 2.5), which can fail if the set of the best matching branches is not a singleton (or it is empty), accordingly to the resolution policy that is used by Java for static overloading. If the best matching branch is found, then we use its signature to annotate the method invocation expression. However, during evaluation, types (both of the receiver and of the parameters) can decrease, then the set of matching branches can increase (remember that, thanks to copy semantics, such set never decreases). Thus, if we simply relied on *minsel* to select the right branch also at run-time, we might not be able to solve diamond relation situations as the one depicted in Fig. 8.

The lookup function *bminsel* uses the (static) type annotation as a top filter to limit the dynamic search, so avoiding all the branches that are completely unrelated to the static choice. Let us stress that this is not a resolution of an ambiguity that gives priority to a given branch w.r.t. another one (similar to some multiple inheritance ambiguity resolutions that automatically choose a class that comes first in the inheritance specification [43,31]). The function *bminsel* selects the most specialized signature by skipping diamond situations; however, the branch chosen at run-time will never be less specialized than the one considered statically during type checking. Thus, the selection is still safe.

On the other hand, one might think of using *bminsel* also in the (static) type checking where Object can be taken as the top filter. While this solution would not affect the safety of the language, it would however be too liberal, accepting as

correct also programs where the ambiguity is clear. For instance, consider the situation depicted in Fig. 8, and the invocation `this.m(new E())`. If we used *bminsel* also in the type system, this method invocation expression would not be rejected; instead, both Java and C++ reject the above expression as ambiguous during compilation (as it is in our typing).

As pointed out in the Introduction, an important feature of our approach consists in preserving Java modularity and program extendibility, so that modules can be type checked in isolation and then composed dynamically without typechecking the whole program again. If one is willing to have a global type checking, he can still enforce the requirement of global well-formedness, according to the approach presented in [9]. In that approach, condition 3 of Definition 2.1 is the weakest condition for ensuring that, for any sequence of input types, the set of matching types (subset of the multi-type) has a least element (see [19], page 119, for the formal proof). Thus, once typing has successfully checked class declarations and method definitions, every method invocation is safe since it can choose this least element, both statically (if the set of matching types is not empty) and dynamically, by simply using the same lookup function *minsel*. It is easy to verify that, if a method `m` is globally well-formed then, in the present solution, for any invocation of `m` we select exactly the least element among all the matching branches. Namely, the type annotation corresponds to this minimum w.r.t. the static types of the arguments. Dynamically, the function *bminsel* returns the minimum w.r.t. the dynamic types of the arguments, and this minimum is a subtype of the static one.

Finally, we want to stress that we are not considering run-time errors, since we are interested in the (type) safe treatment of multi-methods. It is, however, obvious that switching to an "unsafe" version of FMJ, with exceptions due to possible run-time ambiguities, is only a matter of using *minsel* also at run-time.

## 6. Related work

Various object-oriented languages have been proposed to study multi-methods and dynamic overloading. In the following we mention most of them, by focusing on their features that differ from our approach.

*CLOS* [12] is a class-based language with a linearization approach to multi-methods: they are ordered by prioritizing argument position with earlier argument positions completely dominating later ones. This automatic handling of ambiguities may lead to programming errors. In *Dylan* [43] methods are not encapsulated in classes but in generic functions which are first class objects. When a generic function is called it finds the methods that are applicable to the arguments and selects the most specific one. *BeCecil* [21] is a prototype-based language with multi-methods. Multi-methods are collected in first-class generic function objects which can extend other objects. Even if this language is object-based, it provides a static type system, scoping and encapsulation of all declarations; however, its approach, being object-based is radically different from our class-based setting.

The language KOOL [19] integrates multi-methods and overloaded functions (external to classes) in a functional kernel object-oriented language in a type safe way. The notion of well-formedness for FMJ types is widely inspired by the one defined in [19]. KOOL includes both encapsulated multi-methods and overloaded functions external to classes (not included in FMJ) thus unifying in a single language two different styles of programming; on the other hand KOOL does not permit mutually recursive class definitions, namely, it does not allow the programmer to write a method with a parameter of type *A* in a class if the class *A* is defined after the current class. Finally, KOOL does not interpret overloading by copy semantics, so many multi-method definitions which are well typed in FMJ are discarded in KOOL. Thus FMJ permits a less restrictive overloading resolution policy still maintaining the primary goal of type safety.

Tuple [34] is a simple object-oriented language designed to integrate dynamic overloading in a language by adding tuples as primitive expressions. Tuple is statically typed and it is proved to be type safe based on the type checking algorithm presented in [20]. In Tuple methods can be defined either inside classes or in *tuple classes* (external to standard classes). Message lookup on methods defined in a tuple supports dynamic overloading: messages are sent to tuples of arguments dynamically involved in method selection. There are some key differences between Tuple and FMJ. Differently from FMJ, Tuple supports dynamic overloading only for methods external to classes.

Dubious [38] is a core calculus designed to study modular static type checking of multi-methods in modules. Dubious is classless and includes multi-methods with symmetric dispatch in the form of generic functions defined in modules that can be checked separately and then linked in a type safe way. In [38] several type systems are discussed in order to find the right balance between flexibility and type safety. Dubious and FMJ share some basic features such as the symmetry of method dispatch and static type safety.

Fortress [3] is an object-oriented language supporting methods within *traits* [42] and functions defined outside traits. It also provides components (units of compilation) which contain declarations of objects, traits and functions. Fortress differs from mainstream languages since it is not class-based.

Parasitic multi-methods [13] are a variant of the encapsulated multi-methods of [14,19] applied to Java. This extension is rather flexible and indeed provides modular dynamic overloading. The goal of modularity has influenced many aspects of the design:

- method selection is asymmetric, i.e., the receiver's type is evaluated before the argument, in method selection;
- the selection of the most specialized method takes place through `instanceof` checks and consequent type casts;
- parasitic methods are complicated by the use of textual order of methods in order to resolve ambiguities for selecting the right branch;

- all methods must be declared in the class of the receiver in order to eliminate class dependencies. The price to pay is that the class hierarchies of the multi-methods arguments must be anticipated limiting flexibility;
- the rules concerning the interaction between standard and parasitic methods w.r.t. overriding and overloading are quite complex thus requiring some extra effort to the programmer.

*MultiJava* [22] is an extension of Java to support multi-methods and open classes, i.e. classes to which new methods can be added without editing the class directly. New methods in a class C can be added by defining external *method families* in a different compilation unit w.r.t. to the one containing C definition. Only clients that import the new external method family will see the new methods in the apparent signature of C. The drawback of this approach is that extending or modifying an existing method, can only be done by explicitly subclassing all affected variants and overriding the corresponding branches. This complicates the extensibility and can lead to an inconsistent distribution of code. MultiJava modular type system is based on the one of Dubious.

In [41] C++ is extended with open multi-methods and symmetric dispatch. Differently from our approach open multi-methods are external to classes (as the external added methods of MultiJava). The solution proposed in [41] is modular and handles call resolution in three stages (overload (1) and ambiguity (2) resolution, run-time dispatch (3)) to manage ambiguities resolution for open multi-methods in a language supporting multiple inheritance.

*Cmm* [45] is a preprocessor providing multi-methods in C++. *Cmm* is based on the proposal of [44]. The drawback of this solution is that exceptions can be raised due to missing branches and ambiguities, thus losing the advantage of having a type-safe linguistic extension.

Other approaches add dynamic overloading to Java without type checking the proposed extension: *JMMF* [26] is a framework implemented using reflection mechanism, while in [15] a new construct is created using ELIDE (a framework implemented to add high level features to Java). The major drawback of these proposals is that type errors, due to missing or ambiguous branches, are caught at run-time by exceptions.

An extension of the JVM is proposed in [25] in order to provide dynamic overloading in Java without modifying neither the syntax nor the type system: the programmer directly selects the classes which should use dynamic overloading. The problem of this approach is that ambiguous method calls can arise dynamically.

A library extension that implements a limited form of dynamic overloading in Java, based on a visitor pattern-like code, is presented in [28]. Again, this approach suffers from possible run-time exceptions.

Alexandrescu (Chapter 11 of [2]) presents some solutions to implement dynamic overloading in C++ through a smart use of generic programming (templates). This approach does not extend the language and run-time errors due to missing branches can still be raised. Moreover, some of the approaches presented in [2] do not correctly interact with inheritance.

## 7. Conclusions and further work

The contribution of this paper is to provide a formal framework for reasoning about extensions of Java-like languages with multi-methods. Distinguishing features of our proposal are: (i) multi-methods are encapsulated in classes, (ii) dynamic selection of the multi-method branch is symmetric, (iii) we adopt a copy semantics of inheritance, (iv) the extended language FMJ preserves type safety. Furthermore, we statically rule out potential dynamic ambiguities by only typechecking multi-method invocation expressions, without requiring to inspect the whole class hierarchy. Thus we avoid any loss of modularity, extensibility and program maintenance, which is the crucial drawback of many other proposals, including our previous solution presented in [9].

In the present formal perspective, we do not deal with efficiency issues: the dynamic overloading selection here presented is not efficient due to the recurrent use of the lookup functions (Fig. 7), which basically inspect the classes in the hierarchy and all the available branches. On the contrary, in the actual implementation of a language, it is crucial that the selection takes place efficiently; for instance, dynamic binding is usually implemented in constant time, i.e., independently from the number of classes of the hierarchy.

Indeed, from the implementation point of view, the lookup functions must be intended as lookup tables that are built once and for all, possibly by the compiler. However, this would still require a run-time inspection of these tables, and the branch selection might not be constant (see, e.g., some works on the optimizations of these tables, such as [5,24,47]). Moreover, concerning the type checking of multi-method signatures with respect to a given list of types, a polynomial-time algorithm is developed in [20] which is applicable to many contexts, including FMJ.

We already did some work on multi-methods as a language extension of C++ (see [7,8]). In particular, the software `doublecpp`, `http://doublecpp.sf.net`, is based on a preprocessor that, given a program written in the extended C++, produces an equivalent program in standard C++, by applying a program transformation based on the double dispatch technique [30,27]. The translated code implements dynamic overloading selection by using only static overloading and dynamic binding (in particular, it never uses RTTI checks) and thus the selection takes place in constant time, independently from the class hierarchy and from the number of branches.

We plan to follow the same technique to extend Java with multi-methods: we will adapt the transformation technique used for C++ to Java, we will formalize the transformation from an extended Java to standard Java using the extension of FJ presented in this paper. The target of the transformation will be FJ with static overloading, as illustrated in Section 4. The

formalization will prove not only that the translated program is still type safe, but also that the semantics of the translated program is the same as the semantics of the original one.

## Acknowledgments

We are very grateful to the anonymous referees for providing many suggestions on how to make the paper clearer.

## References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer, 1996.
[2] A. Alexandrescu, Modern C++ Design, Generic Programming and Design Patterns Applied, Addison Wesley, 2001.
[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstadt, The Fortress language specification Version 1.0, Sun Microsystems, available on line, 2006.
[4] D. Ancona, S. Drossopoulou, E. Zucca, Overloading and Inheritance, FOOL 8, 2001.
[5] P. André, J.-C. Royer, Optimizing method search with lookup caches and incremental coloring, in: Proc. of OOPSLA '92, ACM Press, New York, NY, USA, 1992, pp. 110–126.
[6] F. Bancilhon, C. Delobel, P. Kanellakis (Eds.), Implementing an Object-Oriented Database System: The Story of $O_2$, Morgan Kaufmann, 1992.
[7] L. Bettini, S. Capecchi, B. Venneri, Translating double-dispatch into single-dispatch, in: Proc. of Int. Workshop on Object-Oriented Developments, WOOD, 2004, in: ENTCS, vol. 138, Elsevier, 2005.
[8] L. Bettini, S. Capecchi, B. Venneri, Double dispatch in C++, Software: Practice and Experience 36 (6) (2006) 581–613.
[9] L. Bettini, S. Capecchi, B. Venneri, Featherweight Java with multi-methods, in: Proc. of PPPJ, Principles and Practice of Programming in Java, vol. 272, ACM Press, 2007, pp. 83–92.
[10] A. Beugnard, Method overloading and overriding cause encapsulation flaw: An experiment on assembly of heterogeneous components, in: SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM Press, 2006, pp. 1424–1428.
[11] D. Beyer, C. Lewerentz, F. Simon, Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems, in: IWSM '00: Proceedings of the 10th International Workshop on New Approaches in Software Measurement, Springer, 2000, pp. 1–17.
[12] D. Bobrow, L. Demichiel, R. Gabriel, S. Keene, G. Kiczales, Common Lisp object system specification, Lisp and Symbolic Computation 1 (3/4) (1989) 245–394.
[13] J. Boyland, G. Castagna, Parasitic methods: Implementation of multi-methods for Java, in: Proc of OOPSLA '97, in: ACM SIGPLAN Notices, vol. 32(10), ACM, 1997, pp. 66–76.
[14] K.B. Bruce, L. Cardelli, G. Castagna, T.H.O. Group, G. Leavens, B.C. Pierce, On binary methods, Theory and Practice of Object Systems 1 (3) (1995) 217–238.
[15] P. Carbonetto, An implementation for multiple dispatch in java using the elide framework.
[16] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys 17 (4) (1985) 471–522.
[17] G. Castagna, Covariance and contravariance: Conflict without a cause, ACM Transactions on Programming Languages and Systems 17 (3) (1995) 431–447.
[18] G. Castagna, A meta-language for typed object-oriented languages, Theoretical Computer Science 151 (2) (1995) 297–352.
[19] G. Castagna, Object-oriented programming: A unified foundation, in: Progress in Theoretical Computer Science, Birkhauser, 1997.
[20] C. Chambers, G. Leavens, Typechecking and modules for multimethods, ACM Transactions on Programming Languages and Systems 17 (6) (1995) 805–843.
[21] C. Chambers, G. Leavens, BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing, in: The 4th Int. Workshop on Foundations of Object-Oriented Languages, FOOL 4, 1996.
[22] C. Clifton, G. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, ACM SIGPLAN Notices 35 (10) (2000) 130–145.
[23] L. DeMichiel, R. Gabriel, The common lisp object system: An overview, in: Proc. ECOOP, in: LNCS, vol. 276, Springer, 1987, pp. 151–170.
[24] K. Driesen, U. Hölzle, Minimizing row displacement dispatch tables, in: Proc. of OOPSLA '95, ACM Press, 1995, pp. 141–155.
[25] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, W. Holst, Multi-dispatch in the Java virtual machine: Design and implementation, in: Proc. of the 6th USENIX Conf. on Object-Oriented Technologies and Systems, COOTS '01, 2001, pp. 77–92.
[26] R. Forax, E. Duris, G. Roussel, Java multi-method framework, in: Int. Conf. on Technology of Object-Oriented Languages and Systems, TOOLS '00, 2000.
[27] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[28] C. Grothoff, Walkabout revisited: The Runabout, in: Proc. of ECOOP, in: LNCS, vol. 2743, Springer, 2003.
[29] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.
[30] D. Ingalls, A simple technique for handling multiple polymorphism, in: Proc. OOPSLA, ACM Press, 1986, pp. 347–349.
[31] S. Keene, Object-Oriented Programming in Common Lisp, Addison-Wesley, 1989.
[32] R. Kumar, V. Agrawal, A. Mangolia, Realization of multimethods in single dispatch object oriented languages, SIGPLAN Notes 40 (5) (2005) 18–27.
[33] D. Lea, Run-time type information and class design, in: Proc. USENIX C++ Technical Conference, USENIX, 1992, pp. 341–347.
[34] G. Leavens, T. Millstein, Multiple dispatch as dispatch on Tuples, in: OOPSLA '98, ACM Press, 1998, pp. 374–387.
[35] S. Lippman, Inside the C++ Object Model, Addison-Wesley, 1996.
[36] B. Meyer, Eiffel: The Language, Prentice-Hall, 1991.
[37] B. Meyer, Overloading vs. Object technology, Journal of Object-Oriented Programming (October/November) (2001) 3–7.
[38] T.D. Millstein, C. Chambers, Modular statically typed multimethods, Information and Computation 175 (1) (2002) 76–118.
[39] W. Mugridge, J. Hamer, J. Hosking, Multi-methods in a statically-typed programming language, in: Proc. ECOOP '91, in: LNCS, vol. 512, Springer, 1991, pp. 307–324.
[40] B.C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, MA, 2002.
[41] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup, Open multi-methods for C++, in: GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering, ACM, 2007, pp. 123–134.
[42] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: Proceedings of European Conference on Object-Oriented Programming, ECOOP'03, in: LNCS, vol. 2743, Springer, 2003, pp. 248–274.
[43] A. Shalit, The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language, Addison-Wesley, Reading, MA, 1997.
[44] J. Smith, Draft proposal for adding Multimethods to C++, Available at: http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1529.html.
[45] J. Smith, Cmm - C++ with Multimethods, Association of C/C++ Users Journal (2001). http://www.op59.net/cmm/readme.html.
[46] B. Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994.
[47] J. Vitek, R.N. Horspool, Compact dispatch tables for dynamically typed object oriented languages, in: CC '96: Proc. of the 6th Int. Conf. on Compiler Construction, in: LNCS, vol. 1060, Springer, 1996, pp. 309–325.